职责链模式

> 使多个对象都有机会处理请求, 从而避免请求的发送者和接受者之间的耦合关系, 将这些对象连成一条链, 并沿着这条链传递该请求, 直到有一个对象处理它为止

实际开发中的职责链模式

假设我们负责一个售卖手机的电商网站，经过分别交纳500元定金和200元定金的两轮预定后（订单已在此时生成），现在已经到了正式购买的阶段。

公司针对支付过定金的用户有一定的优惠政策。在正式购买后，已经支付过500元定金的用户会收到100元的商城优惠券，200元定金的用户可以收到50元的优惠券，而之前没有支付定金的用户只能进入普通购买模式，也就是没有优惠券，且在库存有限的情况下不一定保证能买到。

我们的订单页面是PHP吐出的模板，在页面加载之初，PHP会传递给页面几个字段。

- orderType：表示订单类型（定金用户或者普通购买用户），code的值为1的时候是500元定金用户，为2的时候是200元定金用户，为3的时候是普通购买用户。
- pay：表示用户是否已经支付定金，值为true或者false，虽然用户已经下过500元定金的订单，但如果他一直没有支付定金，现在只能降级进入普通购买模式。
- stock：表示当前用于普通购买的手机库存数量，已经支付过500元或者200元定金的用户不受此限制。

下面我们把这个流程写成代码：

```
var order = function( orderType, pay, stock ){
    if ( orderType === 1 ){ // 500元定金购买模式
        if ( pay === true ){ // 已支付定金
            console.log( '500元定金预购, 得到100优惠券' );
        }else{ // 未支付定金，降级到普通购买模式
            if ( stock > 0 ){ // 用于普通购买的手机还有库存
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    }
    else if ( orderType === 2 ){ // 200元定金购买模式
        if ( pay === true ){
            console.log( '200元定金预购, 得到50优惠券' );
        }else{
            if ( stock > 0 ){
                console.log( '普通购买, 无优惠券' );
            }else{
                console.log( '手机库存不足' );
            }
        }
    }
    else if ( orderType === 3 ){
        if ( stock > 0 ){
            console.log( '普通购买, 无优惠券' );
        }else{
            console.log( '手机库存不足' );
        }
    }
```

```
};
order( 1 , true, 500); // 输出： 500元定金预购，得到100优惠券
```

用职责链模式重构代码

现在我们采用职责链模式重构这段代码，先把500元订单、200元订单以及普通购买分成3个函数。

接下来把orderType、pay、stock这3个字段当作参数传递给500元订单函数，如果该函数不符合处理条件，则把这个请求传递给后面的200元订单函数，如果200元订单函数依然不能处理该请求，则继续传递请求给普通购买函数，代码如下：

```
// 500元订单
var order500 = function( orderType, pay, stock ){
    if ( orderType === 1 && pay === true ){
        console.log( '500元定金预购，得到100优惠券' );
    }else{
        order200( orderType, pay, stock ); // 将请求传递给200元订单
    }
};
// 200元订单
var order200 = function( orderType, pay, stock ){
    if ( orderType === 2 && pay === true ){
        console.log( '200元定金预购，得到50优惠券' );
    }else{
        orderNormal( orderType, pay, stock ); // 将请求传递给普通订单
    }
};
// 普通购买订单
var orderNormal = function( orderType, pay, stock ){
    if ( stock > 0 ){
        console.log( '普通购买，无优惠券' );
    }else{
        console.log( '手机库存不足' );
    }
};
// 测试结果：
order500( 1 , true, 500); // 输出：500元定金预购，得到100优惠券
order500( 1, false, 500 ); // 输出：普通购买，无优惠券
order500( 2, true, 500 ); // 输出：200元定金预购，得到50优惠券
order500( 3, false, 500 ); // 输出：普通购买，无优惠券
order500( 3, false, 0 ); // 输出：手机库存不足
```

用AOP实现职责链

在之前的职责链实现中，我们利用了一个Chain类来把普通函数包装成职责链的节点。其实利用JavaScript的函数式特性，有一种更加方便的方法来创建职责链。

下面我们改写一下Function.prototype.after函数，使得第一个函数返回'nextSuccessor'时，将请求继续传递给下一个函数，无论是返回字符串'nextSuccessor'或者false都只是一个约定，当然在这里我们也可以让函数返回false表示传递请求，选择'nextSuccessor'字符串是因为它看起来更能表达我们的目的，代码如下：

```
Function.prototype.after = function( fn ){
    var self = this;
    return function(){
        var ret = self.apply( this, arguments );
        if ( ret === 'nextSuccessor' ){
            return fn.apply( this, arguments );
        }
        return ret;
    }
};
var order = order500yuan.after( order200yuan ).after( orderNormal );
order( 1, true, 500 ); // 输出：500元定金预购，得到100优惠券
order( 2, true, 500 ); // 输出：200元定金预购，得到50优惠券
order( 1, false, 500 ); // 输出：普通购买，无优惠券
```